MATEUS FELIPE DE CÁSSIO FERREIRA - GRR20176123

RELATRÓRIO DO DESEMPENHO DE ALGORITMOS PARA MULTIPLICAÇÃO DE MATRIZES POR VETORES E RESOLUÇÃO DE SISTEMAS LINEARES USANDO O LIKWID

(*versão pré-defesa, compilada em 2 de junho de 2019*)

Trabalho apresentado como requisito parcial à conclusão da disciplina CI164 - Introdução à Computação Científica no Curso de Bacharelado em Informática Biomédica, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Informática Biomédica*.

Orientador: Daniel Weingaertner.

CURITIBA PR

2019

RESUMO

O presente trabalho busca apresentar a metodologia utilizada na avaliação de *performance* de dois conjuntos de algoritmos utilizados frequentemente no ambiente da computação científica. É apresentado a arquitetura da máquina utilizada nos experimentos bem como a sequência de testes realizados para tornar o experimento reprodutível em outros contextos. Para que esse experimento fosse realizado, foi necessária a utilização ferramentas de avaliação de performance, como o LIKWID e uma função para medição do tempo de execução. Foi constatado que os algoritmos analisados não utilizam o conjunto de instruções AVX, o que os tornam ineficientes quando comparados com algoritmos semelhantes que utilizam esse conjunto. Notou-se, também, que a taxa de *cache miss* e largura de banda do *cache* L3 estavam consideravelmente altos, o que prejudica a *performance* desses algoritmos, dentro da Computação Científica, quando aplicados a matrizes de ordem superior a 10.000.

Palavras-chave: Avaliação de *performance* 1. Computação Científica 2. LIKWID 3.

ABSTRACT

The present paper seeks to present the methodology used in the performance evaluation of two sets of algorithms frequently used in the scientific computing environment. The architecture of the machine used in the experiments is presented, as well as the sequence of tests performed to make the experiment reproducible in other contexts. For this experiment to be performed, it was necessary to use performance evaluation tools, such as LIKWID and a function to measure the time execution. It was found that the algorithms analyzed do not use the set of AVX instructions, which makes them inefficient when compared with similar algorithms that use this set. It was also noted that the cache miss rate and L3 cache bandwidth were considerably high, which impairs the performance of these algorithms on Scientific Computing, when applied to arrays of order higher than 10,000.

Keywords: Performance Evaluation 1. Scientific Computing 2. LIKWID 3.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Existem diversos problemas que despertam o interesse da comunidade científica, seja pela relevância teórica e prática, ou em decorrência de que soluções que ainda não foram encontradas ou sugeridas de uma maneira mais eficiente MORAES JUNIOR (2017). No cenário da Computação Científica, para que uma solução proposta, utilizando algoritmos, possa ser considerada eficiente, devem ser considerados alguns aspectos, como: memória requerida, tempo de execução e aproveitamento da arquitetura interna da máquina e do processador.

Dessa forma, com o objetivo de avaliar o nível de qualidade das soluções baseadas em algoritmos, torna-se imprescindível analisar o desempenho dessa solução com relação a alguns parâmetros pré-estabelecidos pelo programador. Este trabalho tem o propósito de apresentar os resultados obtidos da análise de *performance* de um conjunto de funções presentes em um algoritmo maior. O primeiro deles tem a finalidade de operacionalizar a multiplicação entre matrizes e vetores de três formas diferentes:

- multiplicação usualmente feita de uma matriz por um vetor;
- multiplicação de uma matriz, implementada como um vetor único (orientado em linhas) por um vetor;
- multiplicação de uma matriz, implementada como um vetor único (orientado em colunas) por um vetor.

Já o segundo algoritmo implementa três métodos para a resolução de Sistemas de Equações Lineares, que são separados entre Métodos Diretos[1] (Eliminação de Gauss) e Métodos Iterativos[2] (Gauss-Jacobi e Gauss-Seidel), além de uma função para calcular a norma L^2 do resíduo da solução encontrada.

O segundo capítulo apresenta as condições em que foram executados os testes de desempenho, não só descrevendo a arquitetura do processador da máquina utilizada nesses testes, mas também apresentando as ferramentas utilizadas para medir a *performance* dos algoritmos.

O terceiro capítulo apresenta a metodologia utilizada nos testes realizados, abordando a maneira com que as ferramentas foram utilizadas para a medição, bem como apresentando a forma correta a ser seguida para que se possa reproduzir os testes realizados utilizando o *script* desenvolvido.

Por fim, quarto capítulo apresenta os resultados obtidos nos testes realizados, exibindo os gráficos de performance dos algoritmos que foram gerados pelo *script*, com sua respectiva

[1] Esse tipo de método conduz à solução exata em um número finito de passos, salvos erros introduzidos pela máquina.

[2] Esse tipo de método baseia-se na construção de sequências de aproximações sucessivas para chegar ao resultado que satisfaça uma condição de parada.

descrição, e analisando o desempenho de cada algoritmo em função dos parâmetros previamente estabelecidos.

2 DESENVOLVIMENTO

A fim de medir o desempenho de um algoritmo, previamente deve-se estabelecer parâmetros para definir o ambiente em que essa avaliação será feita, bem como as ferramentas que serão utilizadas no processo. Este capítulo apresentará a arquitetura do processador empregado nos testes e as ferramentas utilizadas, com o intuito de tornar a avaliação desses algoritmos reprodutível, desde que sejam satisfeitas as mesmas condições apresentadas neste trabalho.

2.1 ARQUITETURA DO PROCESSADOR

O processador escolhido para produzir os experimentos é um Intel Core i5 7500 do tipo *Kaby Lake* e conta com uma frequência de *clock* de 3.4GHz. Esse processador conta com 4 cores tendo uma *thread* por core em um único *socket*. A Figura 2.1 apresenta a tipologia gráfica do único *socket* desse processador que foi extraída do software LIKWID.

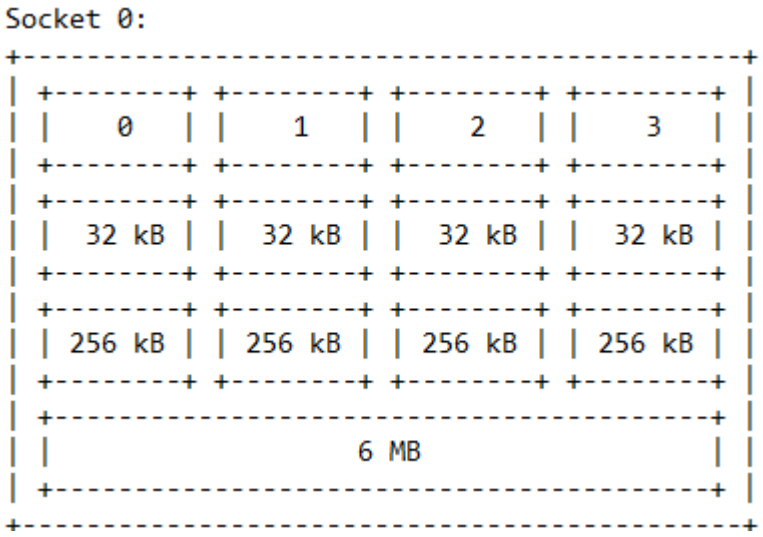

Figura 2.1: *Socket* onde está situado o processador.

Ele conta com uma *cache* L1 não compartilhada de 32 kB e outra *cache* L2 de 256 kB. A *cache* L3, por sua vez, é compartilhada e apresenta um tamanho de 6 MB. A Figura 2.2 apresenta a topologia da cache que também foi extraída por esse software.

A Figura 2.3 Anonymous (2017) extraída do CPU-Z BENCHMARK resume as características desse processador.

```
Level:                  1
Size:                   32 kB
Type:                   Data cache
Associativity:          8
Number of sets:         64
Cache line size:        64
Cache type:             Non Inclusive
Shared by threads:      1
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------
Level:                  2
Size:                   256 kB
Type:                   Unified cache
Associativity:          4
Number of sets:         1024
Cache line size:        64
Cache type:             Non Inclusive
Shared by threads:      1
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 )
--------------------------------------------------------
Level:                  3
Size:                   6 MB
Type:                   Unified cache
Associativity:          12
Number of sets:         8192
Cache line size:        64
Cache type:             Inclusive
Shared by threads:      4
Cache groups:           ( 0 1 2 3 )
```

Figura 2.2: Topologia das *caches* presentes na máquina.

2.2 FERRAMENTAS UTILIZADAS

2.2.1 LIKWID

O LIKWID é uma ferramenta de avaliação de desempenho para ambientes de processadores *multicore* x86 disponível para o sistema operacional GNU Linux. Esse pacote de *software* inclui diversas ferramentas, dentre as quais duas foram utilizadas na construção deste trabalho, como:

- `likwid-topology`: ferramenta utilizada para exibir a topologia da máquina, apresentando a topologia das *threads* de *hardware*, da *cache* e uma topologia gráfica do *socket* da CPU.
- `likwid-perfctr`: ferramenta utilizada para medir os contadores de desempenho de *hardware* em processadores Intel e AMD.

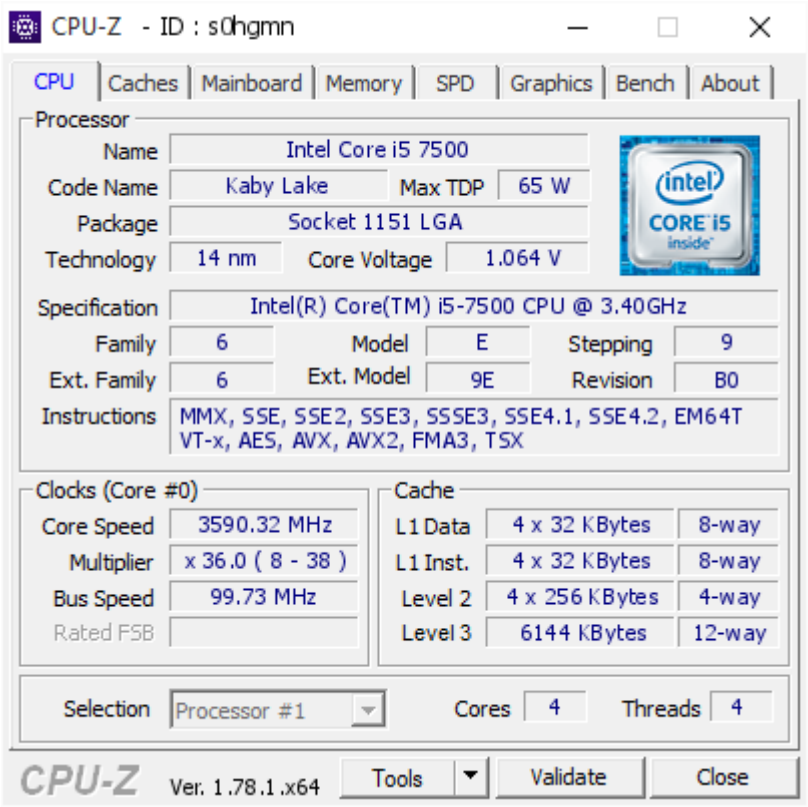

Figura 2.3: Características gerais do processador.

Neste trabalho, a máquina utilizada para realização dos testes dispunha dos indicadores de performance apresentados na Figura 2.4.

Desses indicadores, este trabalho utilizou os grupos: FLOPS_DP, L2CACHE e L3.

2.2.2 TIMESTAMP ( )

Essa função é utilizada para medir o tempo decorrido de uma função, medindo a diferença do tempo de relógio antes e depois na região de interesse. Ela retorna um tipo *double* em milissegundos do tempo medido. O código-fonte para essa função é apresentado abaixo.

```
1 #include <sys/time.h>
2 double timestamp (void);
3
4 double timestamp (void)
5 {
6   struct timeval tp;
7   gettimeofday(&tp, NULL);
8   return((double)(tp.tv_sec*1000.0 + tp.tv_usec/1000.0));
9 }
```

```
Group name      Description
--------------------------------------------------------------
     ICACHE     Instruction cache miss rate/ratio
 PORT_USAGE     Execution port utilization
   FLOPS_SP     Single Precision MFLOP/s
  TLB_INSTR     L1 Instruction TLB miss rate/ratio
   TLB_DATA     L2 data TLB miss rate/ratio
 UOPS_ISSUE     UOPs issueing
UOPS_RETIRE     UOPs retirement
   FLOPS_DP     Double Precision MFLOP/s
    L2CACHE     L2 cache miss rate/ratio
CYCLE_ACTIVITY  Cycle Activities
      CLOCK     Power and Energy consumption
  FLOPS_AVX     Packed AVX MFLOP/s
         L3     L3 cache bandwidth in MBytes/s
FALSE_SHARE     False sharing
  UOPS_EXEC     UOPs execution
    L3CACHE     L3 cache miss rate/ratio
         L2     L2 cache bandwidth in MBytes/s
       DATA     Load to store ratio
   RECOVERY     Recovery duration
     BRANCH     Branch prediction miss rate/ratio
       UOPS     UOPs execution info
     ENERGY     Power and Energy consumption
```

Figura 2.4: Lista de indicadores de *performance* disponíveis para essa máquina.

3 METODOLOGIA

A primeira parte da metodologia consistiu em determinar o conjunto de algoritmos que serão testados durante as avaliações de performance. O primeiro deles está contido no programa `matmult`, que basicamente efetua diferentes maneiras de multiplicação de uma matriz por um vetor. O segundo engloba algoritmos para calcular a resolução de Sistemas de Equações Lineares e está no programa `SistemasLineares`.

Já a segunda parte, baseou-se em separar os testes realizados em dois grupos:

- o teste de tempo de execução, utilizando a função timestamp ( );
- o grupo de testes que não dependiam de tempo (L3, CACHEL2 e FLOPS_DP), utilizando o LIKWID.

Essa separação foi importante para evitar que a medição do tempo de execução pudesse sofrer distorções por conta dos outros testes realizados nos algoritmos. Sendo assim, para os dois programas que serão testados existem marcadores específicos para determinar o tempo de execução de cada função. A Figura 3.1 mostra os marcadores utilizados no programa `matmult` para o cálculo do tempo. Já a Figura 3.2 mostra os marcadores utilizados no programa `SistemasLineares`.

```
printf("%d  ", n);
tempo = timestamp();
multMatPtrVet (mPtr, vet, n, n, resPtr);
tempo  = timestamp() - tempo;
printf("%.5g  ",tempo);


tempo = timestamp();
multMatRowVet (mRow, vet, n, n, resRow);
tempo  = timestamp() - tempo;
printf("%.5g  ",tempo);


tempo = timestamp();
multMatColVet (mCol, vet, n, n, resCol);
tempo  = timestamp() - tempo;
printf("%.5g  \n",tempo);
```

Figura 3.1: Marcadores de tempo utilizados no programa matmult.

É importante notar que a Figura 3.2 apresenta uma forma diferente de obtenção do tempo de execução das funções quando comparada com a Figura 3.1. Isso se deve pelo fato de

```
tempoG = 0.0;
conta_entrada = 0;

printf("%d  ", n);
tempoL = timestamp();
verificador = eliminacaoGauss (SL, &normaL2);
tempoL  = timestamp() - tempoL - tempoG;
printf("%.8g  ",tempoL);
tempoG = 0.0;

tempoL = timestamp();
verificador = gaussJacobi (SL, EPS, &normaL2, &iter);
tempoL  = timestamp() - tempoL - tempoG;
printf("%.8g  ",tempoL);
tempoG = 0.0;

tempoL = timestamp();
verificador = gaussSeidel (SL, EPS, &normaL2, &iter);
tempoL  = timestamp() - tempoL - tempoG;
printf("%.8g  ",tempoL);
tempoG = 0.0;

//medição da média do tempo total da normaL2Residuo
printf("%.8g  \n",tempoNormaL2/conta_entrada);
```

Figura 3.2: Marcadores de tempo utilizados no programa SistemasLineares.

que dentro das funções `eliminacaoGauss`, `gaussJacobi` e `gaussSeidel` elas podem chamar a função `normaL2Residuo`. Sendo assim, é necessário remover o tempo gasto pela função do cálculo da norma quando essa é chamada por outras funções a fim de manter a corretude das medições das outras funções e não permitir que se meça um tempo a mais do que o esperado. Ao final do processo, é feito uma média do tempo gasto pela função do cálculo da norma baseado no número de vezes que essa função foi chamada ao longo de todo o algoritmo. Ou seja, se essa função foi chamada uma única vez dentro da `eliminacaoGauss`, o tempo total gasto por essa função é impresso. Caso essa função também seja chamada pelas funções de `gaussJacobi` e `gaussSeidel`, como mostrado na Figura 3.3, é feito uma média do tempo de execução dessa função antes de ser impresso.

Para o outro agrupamento de testes, os grupos L3, L2CACHE e FLOPS_DP utilizaram uma ferramenta de marcador disponível no LIKWID que permite delimitar uma região de interesse no algoritmo a fim de avaliar o desempenho daquela região para diferentes parâmetros. A Figura 3.4 mostra os marcadores utilizados no programa `matmult` e a Figura 3.5 mostra os marcadores utilizados no programa `SistemasLineares`.

É importante ressaltar que, apesar de ter sido feita uma separação entre dois grupos maiores de testes, ambos os grupos, para programa, são compilados com a *flag* avx, o que permite a inclusão de de *flags* de compilação, como -O3, -mavx e -march=native.

```
//CONDIÇÃO DE PARADA DO MÉTODO
norma = normaMAX (x_next, SLc->x, n);
if (norma < eps)
{
    copiaVetor (x_next, SL->x, n);

    tempoG = timestamp ();
    *normaL2 = normaL2Residuo (SL);
    tempoG = timestamp () - tempoG;
    tempoNormaL2  += tempoG;
    conta_entrada ++;

    return 0;
}
```

Figura 3.3: Cálculo do tempo de execução da norma dentro de uma função do conjunto dos Métodos Iterativos.

```
LIKWID_MARKER_INIT;

LIKWID_MARKER_START("multMatPtrVet");
multMatPtrVet (mPtr, vet, n, n, resPtr);
LIKWID_MARKER_STOP("multMatPtrVet");

LIKWID_MARKER_START("multMatRowVet");
multMatRowVet (mRow, vet, n, n, resRow);
LIKWID_MARKER_STOP("multMatRowVet");

LIKWID_MARKER_START("multMatColVet");
multMatColVet (mCol, vet, n, n, resCol);
LIKWID_MARKER_STOP("multMatColVet");

LIKWID_MARKER_CLOSE;
```

Figura 3.4: Marcadores do LIKWID utilizados no programa matmult.

3.1 *SCRIPT*

Com o propósito de automatizar a execução dos testes, foi criado um *script* em *shell* para que sejam realizados todos os testes para determinado programa. Simplificadamente, o código-fonte do *script* pode ser separado em dois grandes blocos de código, como mostrado na Figura 3.6.

Esse *script* permite definir a ordem das matrizes que serão utilizadas para efetuar os testes, bem como o tamanho da imagem do gráfico gerado e a escala de valores no eixo das abscissas.

A ideia central desse algoritmo de teste é direcionar a impressão do resultado dos testes, seja da medição do tempo de execução, seja dos outros grupos de teste, para um arquivo temporário. Esse arquivo temporário será utilizado pelo programa do `gnuplot` para a construção

```
LIKWID_MARKER_INIT;

LIKWID_MARKER_START ("eliminacaoGauss");
verificador = eliminacaoGauss (SL, &normaL2);
LIKWID_MARKER_STOP ("eliminacaoGauss");


LIKWID_MARKER_START("gaussJacobi");
verificador = gaussJacobi (SL, EPS, &normaL2, &iter);
LIKWID_MARKER_STOP ("gaussJacobi");


LIKWID_MARKER_START("gaussSeidel");
verificador = gaussSeidel (SL, EPS, &normaL2, &iter);
LIKWID_MARKER_STOP ("gaussSeidel");

LIKWID_MARKER_CLOSE;
```

Figura 3.5: Marcadores do LIKWID utilizados no programa SistemasLineares.

```
#PARÂMETRO $1 = CORE
#PARÂMETRO $2 = PROGRAMA
LIKWID_CMD="likwid-perfctr -C $1"

#Ordem das matrizes que serão efetuados os testes para ambos os métodos:
elements=(32 50 64 100 128 200 256 300 400 512 1000 1024 2000 2048 3000 4000 4096 5000 10000)

#Escala de valores em x
x_range="[10:10000]"

#Resolução de saída dos gráficos gerados pelo gnuplot.
image_size="1600,900" #1600x900

#TESTE DE PERFORMANCE AUTOMATIZADO PARA O PROGRAMA: matmult.
if [[ "$2" == "matmult" ]]; then
    #execução dos testes de geração dos gráficos
    # ...

#TESTE DE PERFORMANCE AUTOMATIZADO PARA O PROGRAMA: SistemasLineares
elif [[ "$2" == "SistemasLineares" ]]; then
    #execução dos testes de geração dos gráficos
    # ...

else
    echo "Os parâmetros nessários são: <CORE DA MÁQUINA A SER UTILIZADO> <PROGRAMA>."
    echo "As opções de programas são: matmult ou SistemasLineares."
    echo "Por favor, entre com os parâmetros corretos para que o script possa efetuar os testes."
fi
```

Figura 3.6: Parte do *script* usado na automação dos testes.

do gráfico de um determinado teste. Em alguns casos foi preciso formatar a saída das funções utilizando os comandos `grep`, `awk` e `tr` para construir uma tabela no arquivo temporário, em que a primeira coluna da tabela representa a ordem da matriz e as demais os resultados obtidos no teste realizado pelas diferentes funções, como exemplificado na Figura 3.7.

É importante ressaltar que esse *script* leva em conta que as variáveis de ambiente do GNU Linux já estão configuradas corretamente para a execução do programa do LIKWID e que a frequência do *clock* do processador já está configurado e fixado no máximo. É possível configurar esses parâmetros com os comandos da Figura 3.8.

```
    for size in ${elements[@]} ;
    do
        ${LIKWID_CMD} -f -g ${group} -m $PROGRAM -n $size > temp.tmp

        printf "$((size))"
        printf "$(grep "$key" temp.tmp | awk -F"|" '{print $3}' | tr "\n" " ")\n"

    done > $group.tmp

    gnuplot <<- EOF
    reset
    set terminal png size ${image_size} enhanced font 'Verdana,12'
    set style data linespoints
    set style fill solid 2.00 border 0

    set style line 1 lc rgb 'orange' lt 1 lw 2 pt 13 ps 1.0
    set style line 2 lc rgb 'red' lt 1 lw 2 pt 7 ps 1.0

    set key font ",10"
    set key horizontal
    set key spacing 3
    set key samplen 3
    set key width 2

    set title 'Medição de desempenho para $group-multMatPtrVet' font ",16"

    set xrange ${x_range}
    set logscale x

    set xlabel "tamanho da matriz (doubles)"
    set xlabel font ",13"
    set ylabel "$magnitude"
    set ylabel font ",13"

    set output "$group-multMatPtrVet.png"
    plot '$group.tmp' using 1:2 with linespoints ls 1 title "DP", \
    '$group.tmp' using 1:3 with linespoints ls 2 title "AVX DP"
EOF
```

Figura 3.7: Formatação das saídas geradas pelos marcadores para a construção da tabela.

```
export PATH=/home/soft/likwid/bin:/home/soft/likwid/sbin:$PATH

export LD_LIBRARY_PATH=/home/soft/likwid/lib:$LD_LIBRARY_PATH

echo "performance" > /sys/devices/system/cpu/cpufreq/policy3/scaling_governor
```

Figura 3.8: Alteração de variáveis de ambiente e aumento do *clock* da máquina.

3.2 *FORMA DE EXECUÇÃO DOS TESTES*

Uma vez que o *script* é capaz de realizar todos os testes apresentados anteriormente de forma automatizada, é necessário apenas fornecer dois parâmetros para o código, como: `./execute.sh <CORE> <PROGRAMA>`, onde

- `CORE`: define o núcleo de processamento a ser utilizado no teste;

- `PROGRAMA`: define o conjunto de algoritmos a ser testado, que pode ser `matmult` ou `SistemasLineares`.

É importante ressaltar que, em consequência do processo de automação de geração dos resultados, não é possível obter os gráficos de testes individualmente para um determinado grupo sem antes comentar partes do código para mudar o fluxo de execução. Sendo assim, uma vez que é lançado o algoritmo, os quatro testes apresentados anteriormente serão executados para todas as funções presentes no programa que foi passado como parâmetro para o *script*.

4 RESULTADOS OBTIDOS

Este capítulo apresenta os resultados obtidos dos quatro testes realizados em cada uma das sete funções apresentadas anteriormente, totalizando um total de vinte e oito gráficos gerados. Esses gráficos foram agrupados em uma única imagem para facilitar a visualização de todos os resultados para uma determinada função.

Após a realização de vários experimentos, nas mesmas condições descritas no Capítulo 2, verificou-se que o tempo médio para a conclusão da análise de desempenho para o programa `matmult` foi de aproximadamente 3 minutos, enquanto que, para que seja feita a análise completa de todos os testes para o programa `SistemasLineares` gastou-se em torno de 35 minutos.

Com o objetivo de comparar melhor os resultados, será feita uma análise por tópicos da seguinte maneira:

- `TEMPO`: representa o tempo gasto, em milissegundos, para cada uma das diferentes ordem de matrizes a que foram submetidos os testes;
- `L3 bandwidth`: representa a quantidade máxima de dados que podem ser transferidos por segundo de uma fonte (memória principal ou *cache*) para o destino (registradores);
- `L2 cache miss ratio`: representa a taxa com que o processador foi buscar um dado na memória principal pois o dado não estava contido na *cache* L2;
- `FLOPS_DP`: representa a quantidade de operações em ponto flutuante de precisão dupla que são realizados, utilizando ou não instruções do tipo AVX[1]

4.1 *MATMULT*

O resultado dos testes para o grupo do `matmult` geraram os seguinte gráficos das Figuras 4.1, 4.2 e 4.3, que foram agrupados.

Na computação, esses tipos de algoritmos são maneiras clássicas de perceber a importância de conhecer o *layout* com que os dados estão armazenados na memória em diferentes tipos de linguagens. Basicamente, há duas maneiras para armazenas *arrays* multidimensionais em memória de forma contígua, como:

- `column major order`: os dados próximos no *array* estão organizados em colunas. Um exemplo de linguagem de programação que utiliza esse tipo de organização é o Fortran.

[1]São extensões da arquitetura do conjunto de instruções x86 para microprocessadores da Intel e da AMD. Essa extensão é capaz de suportar dezesseis registradores (8 números flutuante de precisão simples de 32 bits ou 4 números de ponto flutuante de precisão dupla de 64 bits) Lomont (2011).

- `row major order`: os dados próximos no *array* estão organizados em linhas. Um exemplo de linguagem de programação que utiliza esse tipo de organização é o C/C++.

Após a observação não só dos algoritmos implementados por esse programa, mas também dos resultados obtidos da análise de *performance* realizados, podemos inferir alguns pontos:

- `TEMPO`: dois algoritmos de multiplicação de matrizes por vetores apresentaram resultados bem semelhantes para esse parâmetro, como as funções multMatPtrVet e multMatRowVet.

- `L3 bandwidth`: o algoritmo que mais soube aproveitar os dados que são trazidos ao carregar uma linha de *cache* foi o multMatRowVet, o que explica o baixo fluxo de dados para esse parâmetro quando comparado aos demais algoritmos. Em contrapartida, o algoritmo que mais precisou de um fluxo de dados para dentro da *cache* foi o multMatColVet, uma vez que a maneira com que esse algoritmo acessa os dados contíguos em um *array* não é o mais recomendado para a linguagem C.

- `L2 cache miss ratio`: nesse parâmetro, os algoritmos do multMatRowVet e multMatPtrVet obtiveram resultados bem semelhantes e bem destoantes quando comparado ao algoritmo do multMatColVet. Isso se dá pelo fato de que, ao fazer acesso *strided* os dados na memória, este algoritmo não aproveita os dados que são trazidos por uma linha inteira para dentro da *cache*, ou seja, as linhas de *cache* são carregadas e despejadas rapidamente, um fenômeno conhecido como *cache trashing*.

- `FLOPS_DP`: nesse parâmetro, o algoritmo que mais sobressaiu, quando comparado aos demais, novamente foi o multMatRowVet. Esse algoritmo conseguiu manter uma taxa quase constante de operações em precisão dupla realizados pelo processador, enquanto que o algoritmo de pior desempenho, o multMatColVet, realizou um número consideravelmente menor de operações por segundo. Isso é explicado pela alta taxa de *cache misses* que esse algoritmo forçou acontecer, e isso afeta consideravelmente o *pipeline* de execução, que precisa desviar a execução para buscar um dado da memória. Em comparação, este último algoritmo não conseguiu manter o *pipeline* de execução cheio quando comparado ao algoritmo de melhor desempenho.

4.2 *SISTEMASLINEARES*

O resultado dos testes para o grupo do `SistemasLineares` geraram os seguintes gráficos das Figuras 4.4, 4.5, 4.6 e 4.7, que foram agrupados.

Na computação científica, esses tipos de algoritmos são muito utilizados no cálculo de Soluções de Sistemas de Equações Lineares. Assim, quanto melhor for o desempenho de um algoritmo quando comparado a um outro, melhor ele será explorado pelos programadores.

Após a observação dos algoritmos implementados por esse programa e dos resultados obtidos da análise de *performance* realizados, podemos inferir alguns pontos:

- `TEMPO`: entre os três algoritmos utilizados para resolver os sistemas de equações lineares, o que obteve um desempenho inferior quando comparado aos demais foi a função eliminacaoGauss que gastou mais de 100.000 milissegundos na maior ordem de matriz. Comparativamente, os resultados das duas outras funções se mantiveram semelhantes.

- `L3 bandwidth`: novamente, entre os algoritmos que foram comparados, a eliminacaoGauss apresenta o pior desempenho em volume de dados necessários para a sua execução. Isso deve ao fato de essa função implementar a *Lookup table*, o que aumenta a quantidade de dados que precisa trafegar pela *cache*. Comparativamente, o algoritmo de gaussJacobi conseguiu aproveitar, de uma forma pouco melhor, a largura de banda do *cache* L3.

- `L2 cache miss ratio`: nesse parâmetro, o algoritmo da Eliminação de Gauss conseguiu aproveitar, um pouco melhor, a linha de cache carregada junto quando a função requisitava um acesso próximo à região. Comparativamente, os resultados das duas outras funções se mantiveram semelhantes.

- `FLOPS_DP`: em função de conseguir aproveitar mais a linha de *cache*, favorecendo que se mantivesse um *pipeline* de execução cheio, o desempenho da eliminacaoGauss foi considerável em termos de operações em ponto flutuante realizadas. Comparativamente com as outras funções, as oscilações causadas nesse gráfico podem estar relacionadas ao processo de busca de um dado na memória que não estava na cache.

A função normaL2Residuo, por sua vez, apresenta um tempo de execução relativamente rápido, uma vez que essa função não é chamada em todas as ocasiões por outras funções. A largura de banda do *cache* L3 diminui gradativamente com o aumento da ordem da matriz, e o mesmo acontece com o *cache miss* L2, e podemos inferir que essa função utiliza, um pouco satisfatório, a linha de *cache* trazida pelo processador. Nesse sentido, percebe-se que a quantidade de operações em ponto flutuante de precisão dupla dessa função é elevado, com alguns picos de queda.

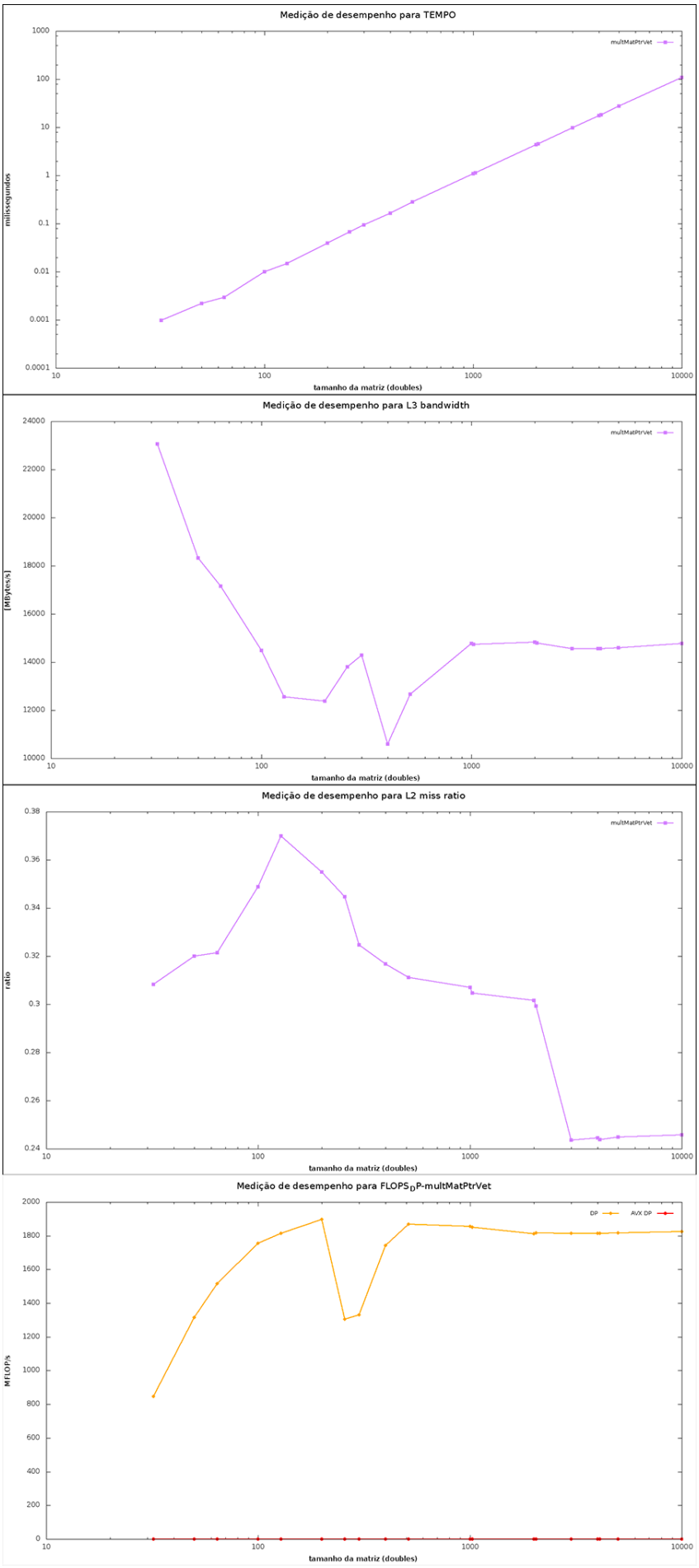

Figura 4.1: Resultados obtidos para a função multMatPtrVet.

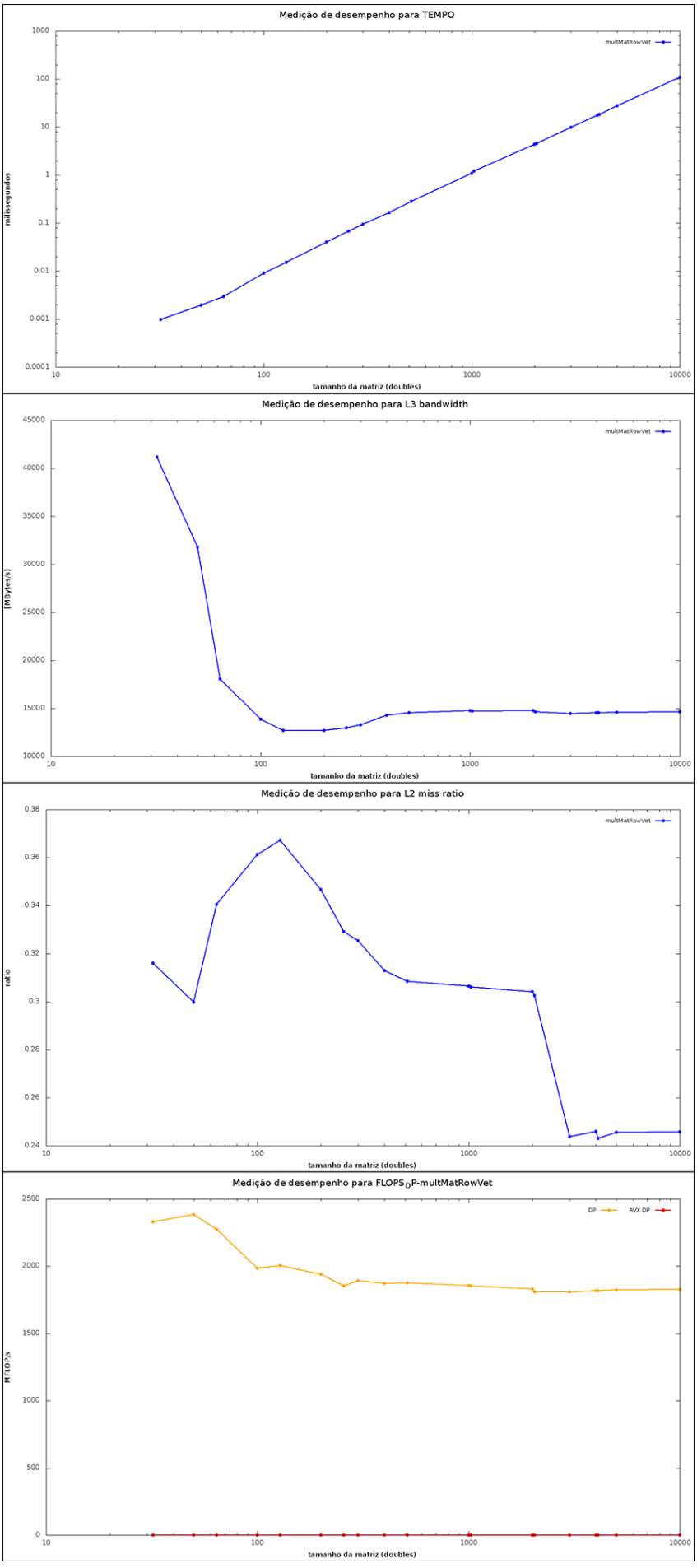

Figura 4.2: Resultados obtidos para a função multMatRowVet.

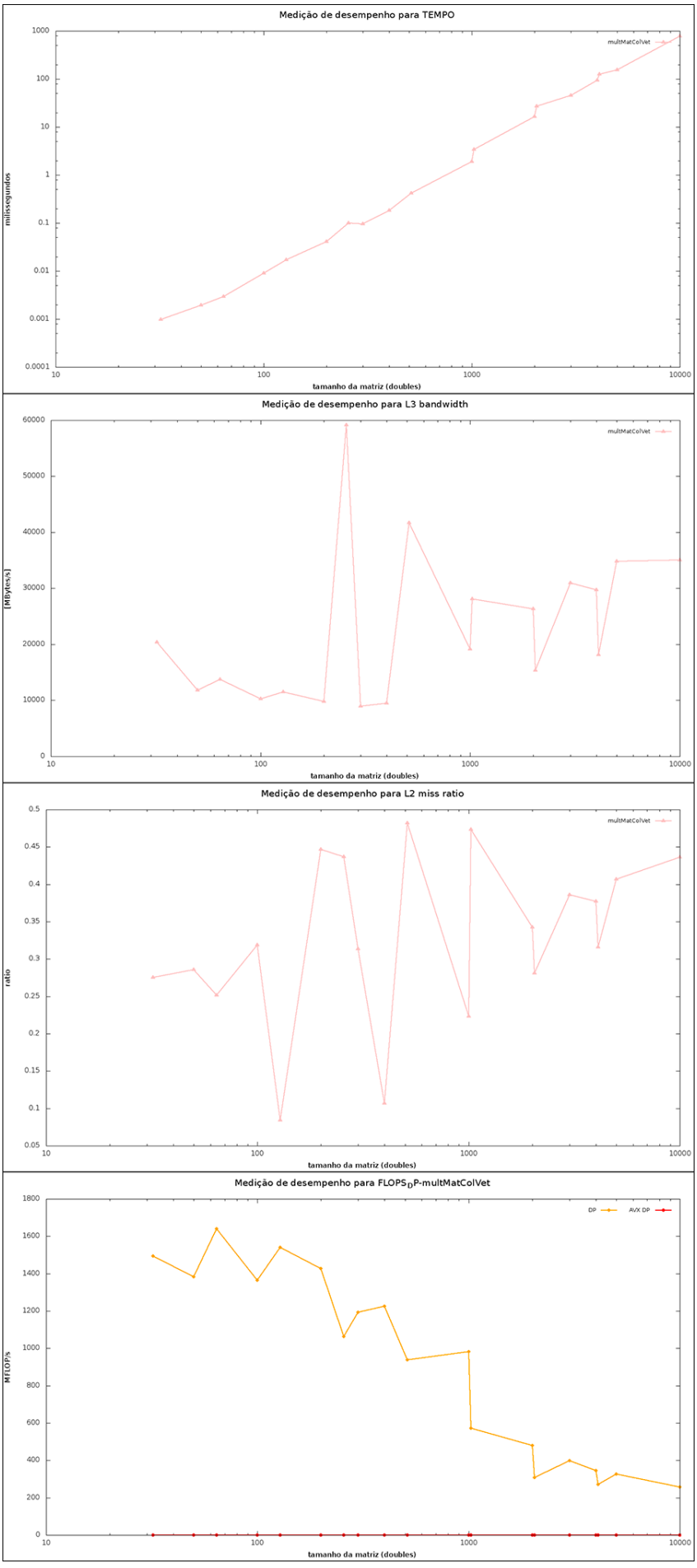

Figura 4.3: Resultados obtidos para a função multMatColVet.

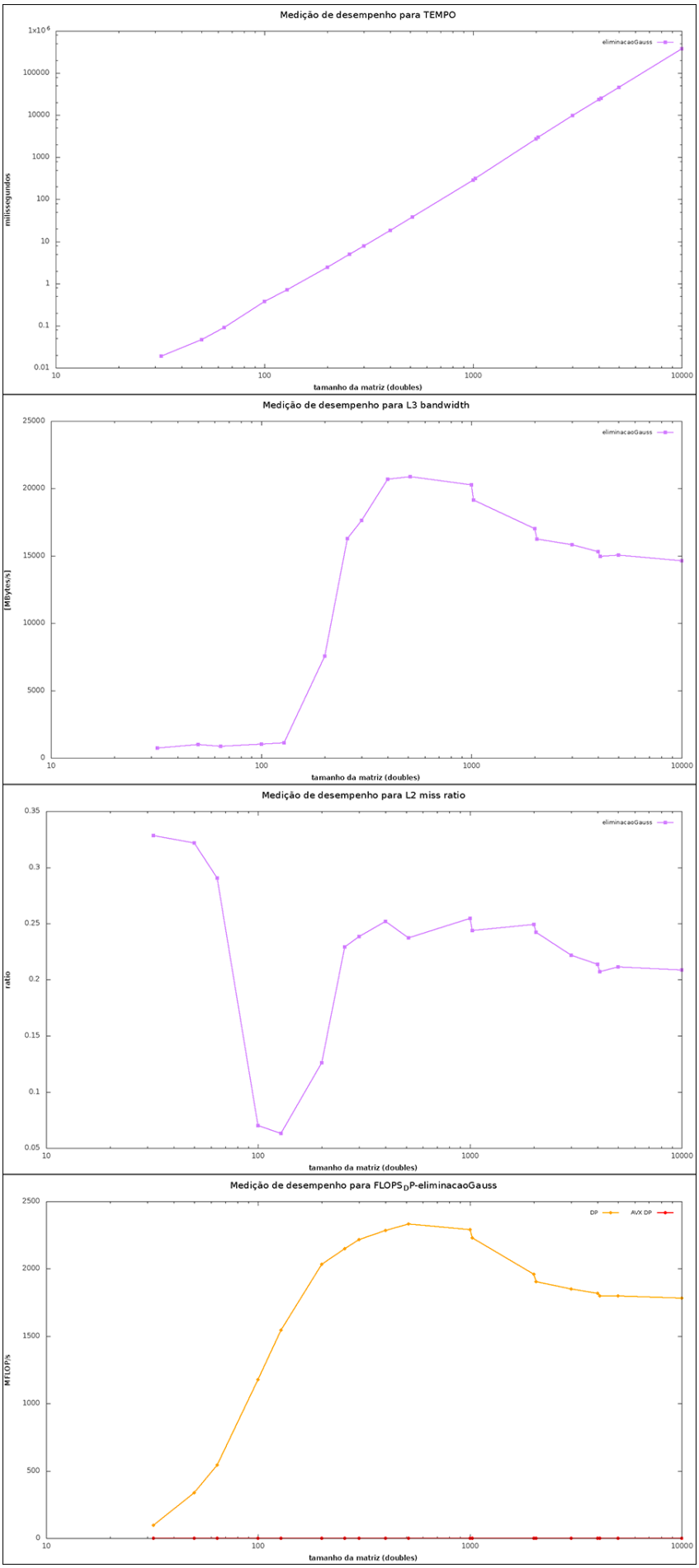

Figura 4.4: Resultados obtidos para a função eliminacaoGauss.

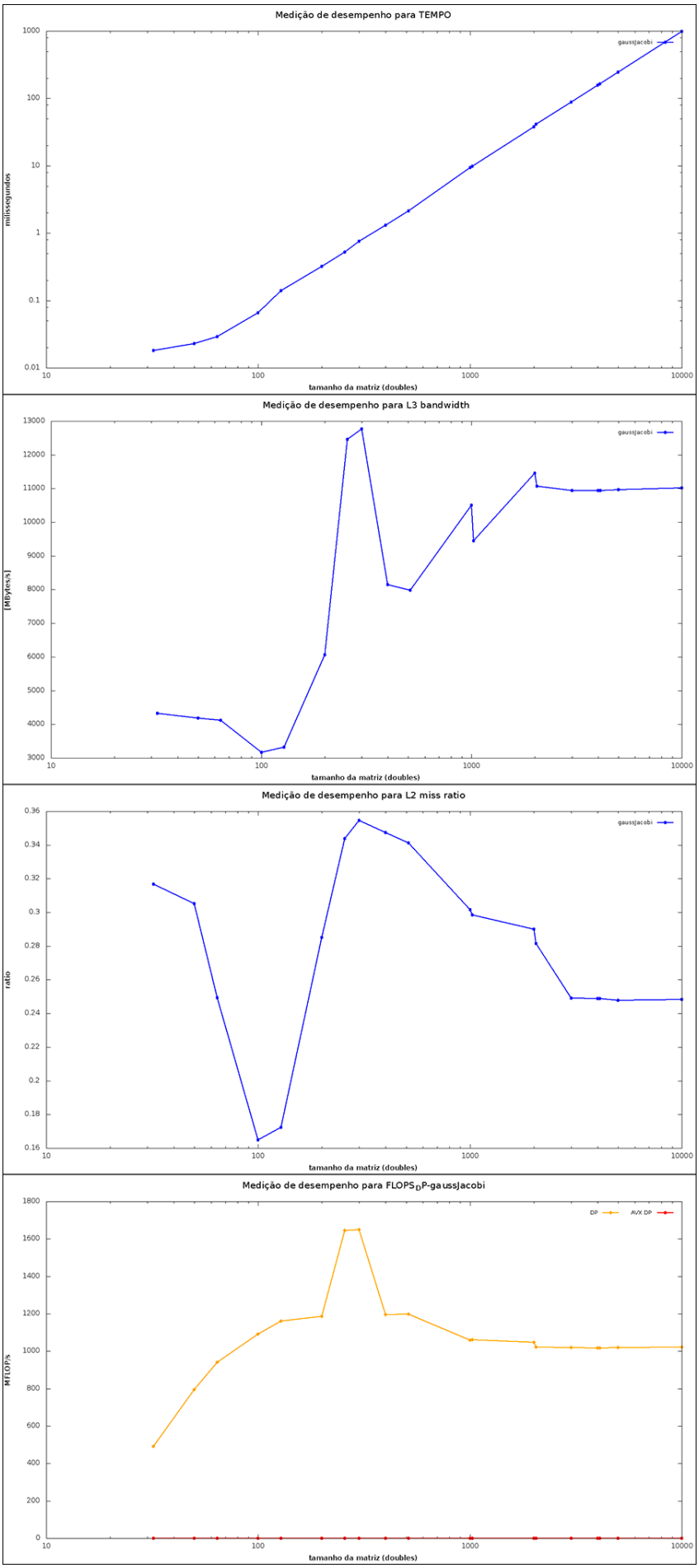

Figura 4.5: Resultados obtidos para a função gaussJacobi.

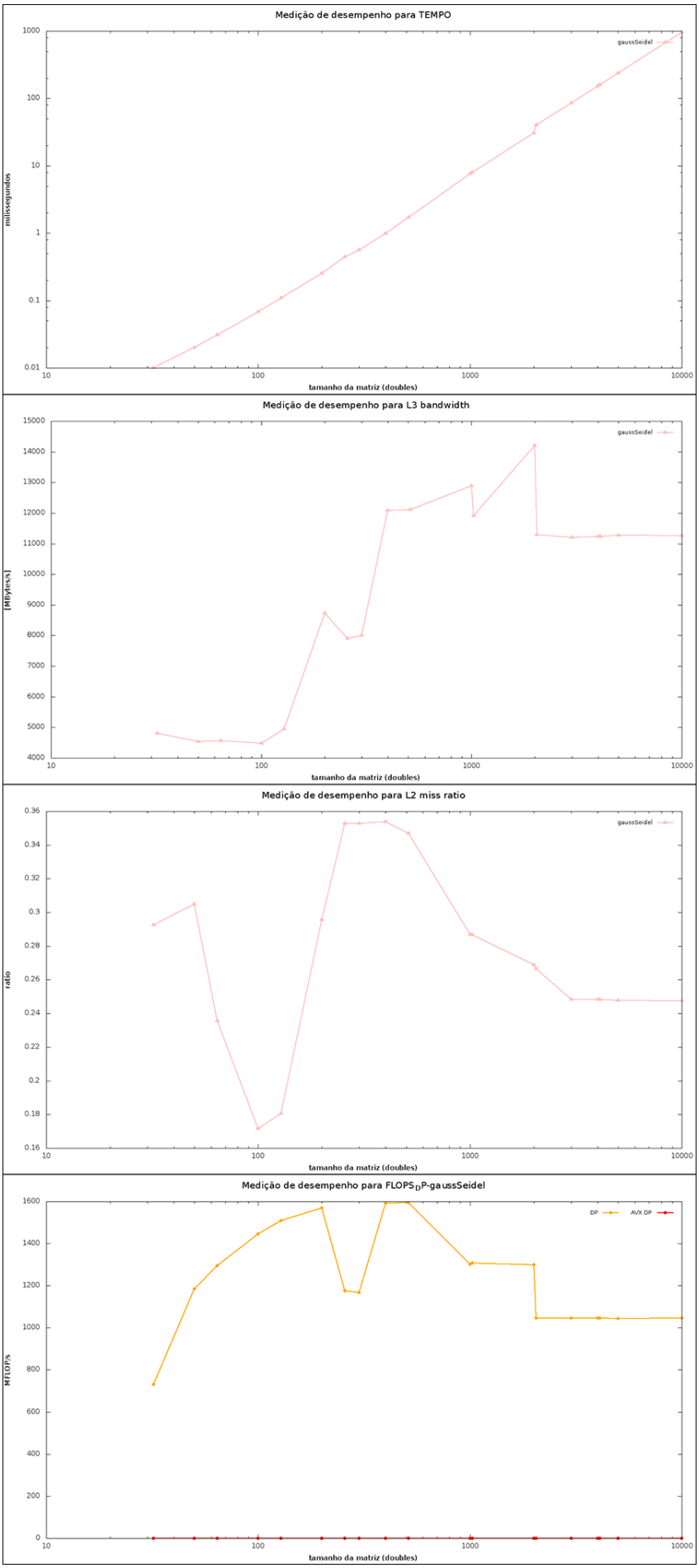

Figura 4.6: Resultados obtidos para a função gaussSeidel.

Figura 4.7: Resultados obtidos para a função normaL2Residuo.

5 CONCLUSÃO

Após o processo de análise de todos os gráficos gerados dos algoritmos que foram testados pelo *script*foi constatado que os algoritmos analisados não utilizam o conjunto de instruções AVX, o que os tornam ineficientes quando comparados com algoritmos semelhantes que utilizam esse conjunto.

Notou-se, também, que a taxa de *cache miss* e largura de banda do *cache* L3 estavam consideravelmente altos, o que prejudica a *performance* desses algoritmos, dentro da Computação Científica, quando aplicados a matrizes de ordem superior a 10.000.

Ainda, é importante ressaltar que o tempo de execução do teste do programa `SistemasLineares` é consideravelmente elevado quando comparado ao algoritmo do `matmult`, embora realizem operações distintas. Esse fator, somado ao fato de as funções implementadas por esse programa não utilizarem de forma satisfatória a *cache* L2, contribui para a baixa *performance* desse algoritmo.

Sendo assim, com o objetivo de aperfeiçoar a execução desses algoritmos, torna-se imprescindível que sejam aplicadas técnicas de otimização de código, principalmente com foco em utilizar o conjunto de instruções AVX que está presente no processador realizado os testes, bem como aproveitar, de maneira satisfatória, o *pipeline* de execução dos algoritmos, evitando a máximo que ocorra uma taxa elevada de *cache miss*.

REFERÊNCIAS

Anonymous (2017). Intel core i5 7500 @ 3590.32 mhz. http://valid.x86.fr/s0hgmn. Acessado em 02/06/2019.

Lomont, C. (2011). Introduction to intel® advanced vector extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. Acessado em 02/06/2019.

MORAES JUNIOR, C. M. (2017). *Uma abordagem para avaliar o desempenho de algoritmos baseada em simulações automáticas de modelos de Petri coloridas hierárquicas*. Tese de doutorado, UFU - Universidade Federal de Uberlândia, Uberlândia - MG. 125 pgs.